POLIMORFISMO

Es la capacidad de los objetos de responder al mismo mensaje en función de los parámetros utilizados durante su invocación o de la clase a la cual pertenece. Se puede invocar a una función de un objeto sin especificar su tipo exacto. Por ejemplo al invocar Figura.Dibujar() se dibujará un círculo, un rectángulo ó un polígono dependiendo del objeto heredado de la clase Figura que llame a la función.

Es la tercera característica esencial de un lenguaje OO (abstracción de datos y herencia).

Permite mejorar la organización del código y su legibilidad así como la creación de programas *extensibles* que pueden "crecer" cuando se deseen nuevas características.

- Asignación estática (**static binding** o **early binding**): la asociación de un objeto con sus operaciones se realiza en tiempo de compilación (sobrecarga de funciones).
- Asignación dinámica (**dynamic binding** o **late binding**): la asociación se realiza en tiempo de ejecución (redefinición de métodos). Las funciones polimórficas se declaran de la misma manera (con el mismo nombre) en **distintas clases** que las definen en forma diferente. Se utilizan conceptos que se encuentran estrechamente relacionados entre sí: **función virtual**, **enlazado dinámico** y **clase polimórfica**.

Métodos virtuales

Constituyen el mecanismo que permite declarar **funciones polimórficas**. Un programa puede invocar, mediante un puntero o una referencia a una clase base, una función definida en una clase derivada (en general diferirán en la implementación).

virtual <**tipo**> <nombre_función>(<lista_parámetros>) [{ }];

Una clase que declara o hereda una función virtual se denomina **clase polimórfica**.

Se utiliza cuando se quiere tener un conjunto de objetos de diferentes clases agrupados en un arreglo y realizar una operación particular sobre cada uno de ellos utilizando el mismo nombre para las funciones que pueden tener distinto código.

Por ejemplo, se quiere realizar un programa que dibuje y calcule el área de distintas figuras geométricas: círculo, cuadrado, triángulo, etc. En la superclase, CFigura, se declararán las propiedades comunes a todos los tipos (colLínea, colFondo, iLados, coorVertices[], etc.) y los métodos que se utilizarían. Para cada tipo de figura se heredará de CFigura una nueva clase que será la encargada de implementar el código correcto para cada uno de los métodos (Dibujar, CalcArea, CalcPerimetro, etc.) en función de sus características particulares.

Métodos virtuales – ejemplo 1

El programa agrupará los distintos tipos de figura en un vector del tipo CFigura donde **cada elemento** es un **puntero** a **distintos objetos**:

```
CFigura *ptrFormas[20];                 // cantidad de figuras en la pantalla

ptrFormas[0] = new Circulo(....)        // nuevo objeto Circulo en un puntero a CFigura
ptrFormas[1] = new Cuadrado(....)       // nuevo objeto Cuadrado en un puntero a CFigura
ptrFormas[2] = new Triangulo(....)      // nuevo objeto Triangulo en un puntero a CFigura
ptrFormas[3] = new Circulo(....)        // otro objeto Circulo

for(int i=0; i<cant_obj, i++)           // itera para cada tipo de figura con el  MISMO VECTOR DE PUNTEROS
{
        ptrFormas[i]->Dibujar();        // invoca a la función Dibujar de tipo de objeto apuntado
        ptrFormas[i]->CalcArea();       // utiliza la versión de CalcArea que se necesita
}
```

El **mismo llamado** a una función hace que se ejecute **código diferente**!!! Para ocultar el comportamiento de la clase base se deben haber declarado las funciones que se espera sean modificadas **virtuales**, lo que obliga a que se utilice la función que concuerde con el contenido del objeto más que con su tipo. El método está definido con el mismo nombre en todas las clases.

Métodos virtuales – ejemplo 2

Si las funciones de las clases base no se declaran virtuales se puede producir un resultado no deseado (no se invoca la función que corresponde):

```
class Persona
{
   public:
      Persona(string n) { nombre = n; }
      void VerNombre() { cout << nombre << endl; }
   protected:
      string nombre;
};
```

```
class Empleado : public Persona
{
   public:
      Empleado(string n) : Persona(n) {}
      void VerNombre() {cout << "Emp: " << nombre << endl; }
};

class Estudiante : public Persona
{
   public:
      Estudiante(string n) : Persona(n) {}
      void VerNombre() { cout << "Est: " << nombre << endl; }
};
```

```
int main()
{
   Persona *Jose = new Estudiante("Jose");
   Persona *Carlos = new Empleado("Carlos");
   Carlos->VerNombre();
   Jose->VerNombre();
   delete Jose;
   delete Carlos;
   return 0;
}
```

```
Salida:   Carlos      // no imprime Emp: Carlos
          Jose        // no imprime Est: Jose
```

Métodos virtuales – ejemplo 2

Para que se ejecute la función VerNombre() de *Estudiante* o *Empleado* dependiendo de la clase de objeto a la que apunte el puntero se debe declarar virtual la de la clase base.

```
class Persona
{
   public:
      Persona(string n) { nombre = n; }
      virtual void VerNombre() { cout << nombre << endl; }
      virtual ~Persona() { ; }          // el destructor es declarado virtual aunque no realiza ninguna sentencia
   protected:
      string nombre;
};
```

Salida: Emp: Carlos // **imprime ok**

Est: Jose // **imprime ok**

Ej. 01: Método virtual

Métodos virtuales – ejemplo 3

En lugar de utilizar punteros se pueden utilizar referencias a los objetos. Las declaraciones y definiciones de las clases utilizadas no se modifican.

```
int main()
{
    Estudiante Jose("Jose");
    Empleado Carlos("Carlos");
    Persona&  rJose = Jose;
    Persona& rCarlos = Carlos;
    rCarlos.VerNombre();
    rJose.VerNombre();
    return 0;
}
```

Salida: Emp: Carlos // **imprime ok**
Est: Jose // **imprime ok**

Ej. 02: Referencias

Métodos virtuales

- El compilador crea un vector oculto de punteros denominado **v–table**. Cada uno de los elementos apunta a la función virtual propia. Si en la clase no se ha definido ningún miembro virtual, el puntero apuntará a la función virtual de su clase base más próxima en la jerarquía hasta encontrar la clase donde se ha definido la función buscada. La tabla se crea al construir el objeto, por lo que los **constructores no podrán ser virtuales**, ya que no se dispone del puntero a la tabla hasta terminar con el constructor.

- Una función declarada como virtual debe ser definida en la clase base que la declara (excepto si la función es virtual pura), y podrá ser empleada aunque no haya ninguna clase derivada. Las funciones virtuales sólo se redefinen cuando una clase derivada necesita modificar el código implementado por su clase base.

- Una vez se declara un método como virtual sigue siéndolo en todas las clases derivadas, la propiedad se hereda. Aunque no es necesario, es recomendable incluir siempre la palabra virtual en las redefiniciones de la función (si existen), para tener presente esa característica en especial en derivaciones múltiples.

- **Tampoco** ser declarados virtuales los métodos estáticos, y los operadores new y delete.

Constructores copia virtuales

Los constructores no pueden ser virtuales, por lo que si se necesitara implementar en las clases derivadas el constructor de copia, se debería crear una función virtual "*clonar*" en la clase base y redefinirla en cada clase derivada.

Destructores virtuales

Si se destruye un objeto referenciado mediante un puntero a su clase base, se invocará el destructor de la misma y no el que se haya definido en la clase derivada. Esto puede ocasionar un comportamiento totalmente indeseado ya que la clase derivada puede tener que realizar un número mayor de tareas. La solución es declarar como virtual el destructor de la superclase, con lo cual si alguna de las clases derivadas lo sobrescribió se ejecutará el código correcto. Se debe respetar siempre: **si en una clase existen funciones virtuales, el destructor debe ser virtual**.

Ej. 03: Constructor copia

Funciones virtuales puras

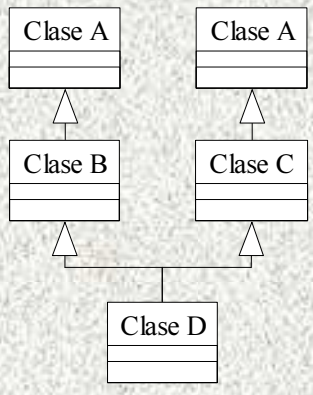
Una función virtual pura es aquella que no necesita ser definida, aunque debe ser declarada. El modo de declararla es asignándole el valor cero.

virtual <**tipo**> <nombre_función>(<lista_parámetros>) [{ }] = 0;

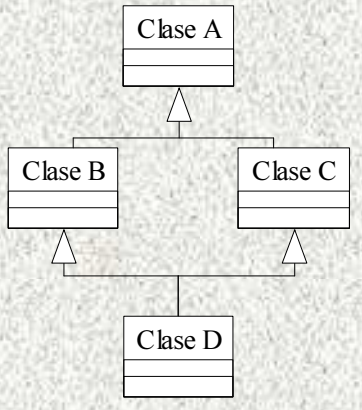
Por ejemplo en la clase CFiguras el método Dibujar() depende de cada una de las formas. Se declara virtual pura en la clase base, lo que obliga a implementar los métodos de dibujo en cada una de las clases derivadas.

Clase base virtuales

Puede suceder que una clase herede indirectamente varias veces los miembros de otra clase. Los miembros de clase *A* se encontrarán duplicados en la clase *D*. por lo que se presenta una ambigüedad a la hora de acceder a los datos o funciones heredades.



Derivación ambigua



Derivación no ambigua

Para evitar este problema las clases *B* y *C* deben derivar de la clase *A* declarándola **clase base virtual**, provocando que los miembros de este tipo se hereden sólo una vez.

class <Clase *B*/*C* > **:** virtual public <Clase *A*> { ... };

class <Clase *D*> **:** public <Clase *B*>, public <Clase *C*> { ..... };

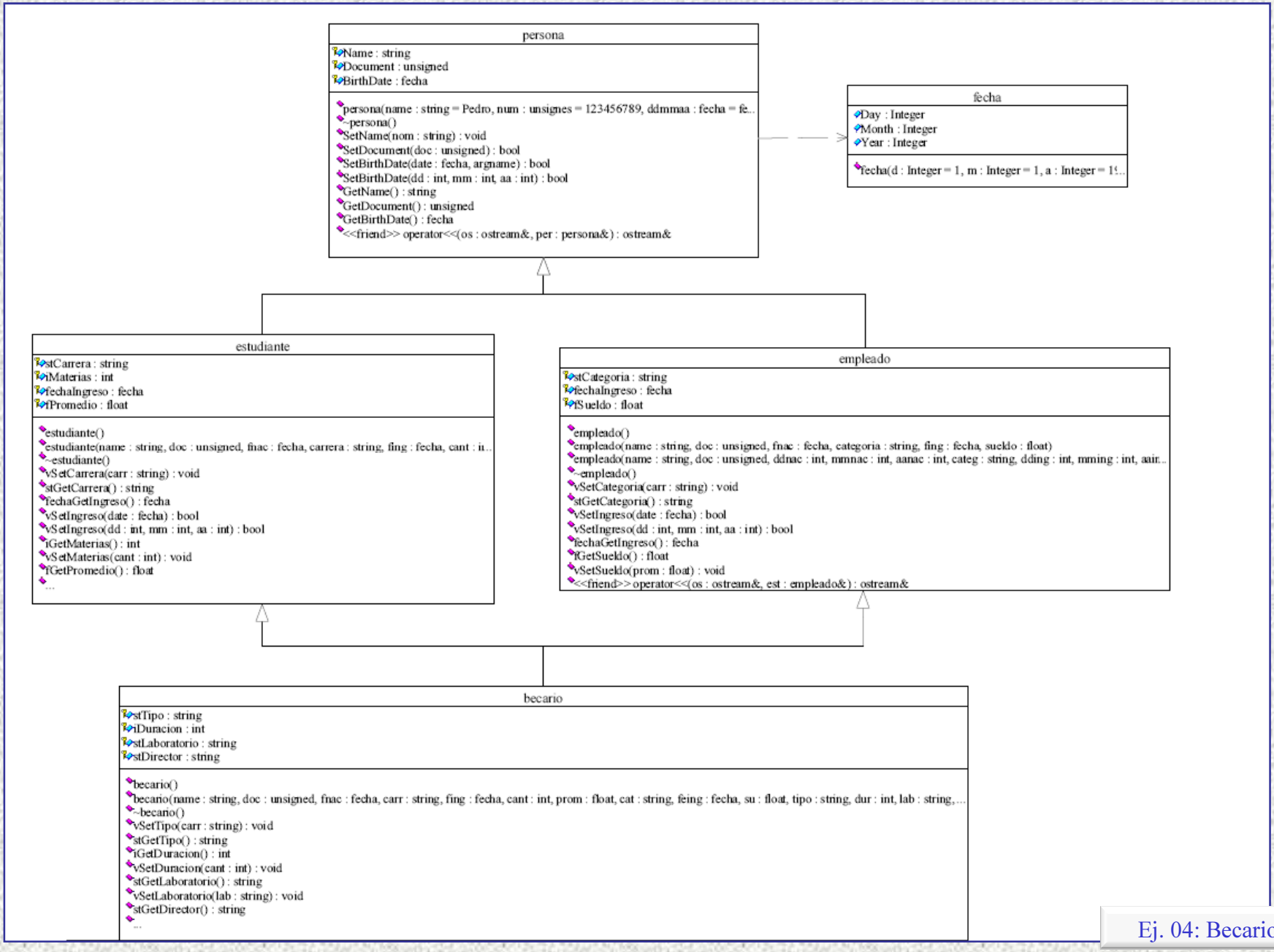
Desde el punto de vista de Clase *B/C* no hay diferencia respecto a la declaración original, en cambio la Clase *D* sólo hereda una vez la Clase *A*.

Clase base virtuales

Se debe considerar que el constructor de la Clase *A* deberá ser invocado desde el de la Clase *D*, ya que ni la Clase *B* ni la Clase *C* lo harán automáticamente. Si no se hace el compilador genera un error.

Si una clase base que se hereda como virtual define constructores, **debe** proporcionar un constructor sin parámetros **ó** un constructor con proporcione todos los valores por defecto. Los constructores de las clases base virtuales son invocados antes que los de cualquier otra clase base no-virtual. Si la jerarquía de clases contiene múltiples clases-base virtuales, sus respectivos constructores son invocados en el mismo orden que fueron declaradas. Después se invocan los constructores de las clases-base no virtuales, y por último el constructor de la clase derivada.

Ejemplo: clase becario modificada



Ej. 04: Becario

Clases abstractas

Se puede definir una clase para que sólo sea un modelo para las derivadas sin que se declare ningún objeto específico. Se las denomina clases base **abstractas** (ABC, Abstract Base Class) y su función es la de agrupar miembros comunes de otras clases que se derivarán de ellas.

Una clase abstracta es aquella que posee al menos una función virtual pura. La declaración de esta función especifica cómo debe ser el mensaje solamente (cantidad y números de parámetros con que se invoca). Si una clase derivada no define una función virtual pura, la hereda como pura y por lo tanto también es abstracta. A menudo se mencionan las clases abstractas como tipos de datos abstractos, en inglés: Abstract Data Type, o resumido **ADT**.

No es posible crear objetos de una clase abstracta, estas clases sólo se usan como clases base para la declaración de clases derivadas.

En UML, se designa una clase como abstracta al poner el nombre de la clase en cursiva.

Características del dynamic binding

- La asociación de un objeto con sus operaciones se hace en tiempo de ejecución.
- Permite modificar el comportamiento de un código compilado. Se puede hacer un módulo nuevo que maneje nuevos tipos sin necesidad de modificar el código fuente y recompilarlo.
- Se implementa en C++ mediante el uso de una tabla de punteros a funciones virtuales (v-table) que el compilador construye para cada clase que usa funciones virtuales.
- El enlace dinámico no es costoso en tiempo de ejecución